Tying up the loose ends in fully LZW-compressed pattern matching*

Paweł Gawrychowski[1]

1 **Institute of Computer Science, University of Wrocław, Poland**
Max-Planck-Institute für Informatik, Saarbrücken, Germany
gawry@cs.uni.wroc.pl

Abstract

We consider a natural generalization of the classical pattern matching problem: given compressed representations of a pattern $p[1..M]$ and a text $t[1..N]$ of sizes m and n, respectively, does p occur in t? We develop an optimal linear time solution for the case when p and t are compressed using the LZW method. This improves the previously known $\mathcal{O}((n+m)\log(n+m))$ time solution of Gąsieniec and Rytter [11], and essentially closes the line of research devoted to studying LZW-compressed exact pattern matching.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases pattern matching, compression, Lempel-Ziv-Welch

Digital Object Identifier 10.4230/LIPIcs.STACS.2012.624

1 Introduction

One of the most natural problems concerning processing information is *pattern matching*, in which we are given a pattern $p[1..M]$ and a text $t[1..N]$, and have to check if there is an occurrence of p in t. Although many very efficient (both from a purely theoretical and more practically oriented point of view) solutions to this problem are known [16, 8, 13, 9, 5, 4], most data is archived and stored in a compressed form. This suggest an intriguing research direction: if the text, or both the pattern and the text, are given in their compressed representations, do we really need to decompress them in order to detect an occurrence? If just the text is compressed, this is the *compressed pattern matching* problem. For Lempel-Ziv-Welch compression, used for example in Unix `compress` utility and GIF images, Amir *et al.* introduced two algorithms with respective time complexities $\mathcal{O}(n+M^2)$ and $\mathcal{O}(n\log M + M)$ [1], where n is the compressed size of the text. The pattern preprocessing time was then improved [14] to get $\mathcal{O}(n+M^{1+\epsilon})$ time complexity. In a recent paper [12] we proved that in fact a $\mathcal{O}(n+M)$ solution is possible, as long as the alphabet consists of integers which can be sorted in linear time. A more general problem is the *fully compressed pattern matching*, where both the text and the pattern are compressed. This problem seems to be substantially more involved than compressed pattern matching, as we cannot afford to perform any preprocessing for every possible prefix/suffix of the pattern, and such preprocessing is a vital ingredient of any efficient pattern matching algorithm known to the author. Nevertheless, Gąsieniec and Rytter [11] developed a $\mathcal{O}((n+m)\log(n+m))$ time algorithm for this problem, where n and m are the compressed sizes of the text and the pattern, respectively.

* Supported by MNiSW grant number N N206 492638, 2010–2012 and START scholarship from FNP.

29th Symposium on Theoretical Aspects of Computer Science (STACS'12).
Editors: Christoph Dürr, Thomas Wilke; pp. 624–635

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper we show that in fact an optimal linear time solution is possible for fully LZW-compressed pattern matching. The starting point of our algorithm is the $\mathcal{O}(n+M)$ time algorithm [12]. Of course we cannot afford to use it directly as M might be of order m^2. Nevertheless, we can apply the method to a few carefully chosen fragments of the pattern. Then, using those fragments we try to prune the set of possible occurrences, and verify them all at once by a combined usage of the so-called PREF function and iterative compression of the (compressed) pattern similar to the method of [11]. The chosen fragments correspond to the most complicated part of the pattern, in a certain sense. If there is no such part, we observe that the pattern is periodic, and a modification of the algorithm from [12] can be applied. To state this modification, and prove its properties, we briefly review the algorithm from [12] in the next section. While the modification itself might seem simple, we would like to point out that it is nontrivial, and we need quite a few additional ideas in order to get the result.

2 Preliminaries

We consider strings over finite alphabet Σ (which consists of integers which can be sorted in linear time, namely $\Sigma = \{1, 2, \ldots, (n+m)^c\}$) given in a Lempel-Ziv-Welch compressed form, where a string is represented as a sequence of *codewords*. Each codeword is either a single letter or a previously occurring codeword concatenated with a single character. This additional character is not given explicitly: we define it as the first character of the next codeword, and initialize the set of codewords to contain all single characters in the very beginning. The resulting compression method enjoys a particularly simple encoding/decoding process, but unfortunately requires outputting at least $\Omega(\sqrt{N})$ codewords (so, for example, we are not able to achieve an exponential compression possible in the general Lempel-Ziv method). Still, its simplicity and good compression ratio achieved on real life instances make it an interesting model to work with. For the rest of the paper we will use LZW when referring to Lempel-Ziv-Welch compression.

We are interested in a variation of the classical pattern matching problem: given a pattern $p[1\,..\,M]$ and a text $t[1\,..\,N]$, does p occur in t? We assume that both p and t are given in LZW compressed forms of size m and n, respectively, and wish to achieve a running time depending on the size of the compressed representation $n+m$, not on the original lengths N and M. If the pattern does occur in the text, we would like to get the position of its first occurrence. We call such problem the *fully compressed pattern matching*.

A closely related problem is the *compressed pattern matching*, where we aim to detect the first occurrence of an uncompressed pattern in a compressed text. In a previous paper [12], we proved that this problem can be solved in deterministic linear time. This $\mathcal{O}(n+M)$ time algorithm will be our starting point. Of course we cannot directly apply it as M might be of order m^2. Nevertheless, a modified version of this solution will be one of our basic building bricks. In order to state the modification and prove its correctness we briefly review the idea behind the original algorithm in the remaining part of this section. First we need a few auxiliary lemmas. A *period* of a word w is an integer $0 < d \leq |w|$ such that $w[i] = w[i+d]$ whenever both letters are defined.

▶ **Lemma 1** (Periodicity lemma). *If both d and d' are periods of w, and $d+d' \leq |w|+\gcd(d,d')$, then $\gcd(d,d')$ is a period as well.*

Using any linear time suffix array construction algorithm and fast LCA queries [2] we get the following.

▶ **Lemma 2.** *Pattern p can be preprocessed in linear time so that given any two fragments $p[i\,..\,i+k]$ and $p[j\,..\,j+k]$ we can find their longest common prefix (suffix) in constant time.*

▶ **Lemma 3.** *Pattern p can be preprocessed in linear time so that given any fragment $p[i\,..\,j]$ we can find its longest prefix which is a suffix of the whole pattern in constant time, assuming we know the (explicit or implicit) vertex corresponding to $p[i\,..\,j]$ in the suffix tree.*

A *border* of a word w is an integer $0 \leq b < |w|$ such that $w[1\,..\,b] = w[|w| - b + 1\,..\,|w|]$. The longest border is usually called the border (similarly, the shortest period is called the period). By applying the preprocessing from the Knuth-Morris-Pratt algorithm for both the pattern and the reversed pattern we get the following.

▶ **Lemma 4.** *Pattern p can be preprocessed in linear time so that we can find the border of each its prefix (suffix) in constant time.*

A *snippet* is any substring (factor) $p[i\,..\,j]$ of the pattern, represented as a pair of integers (i, j). If $i = 1$ we call it a *prefix snippet*, and if $j = |p|$ a *suffix snippet*. An *extended snippet* is a snippet for which we also store the corresponding vertex in the suffix tree (built for the pattern) and the longest suffix which is a prefix of the pattern. A sequence of snippets is a concatenation of a number of substrings of the pattern.

A high-level idea behind the linear time LZW-compressed pattern matching is to first reduce the problem to pattern matching in a sequence of extended snippets. It turns out that if the alphabet is of constant size, the reduction can be almost trivially performed in linear time, and for polynomial size integer alphabets we can apply more sophisticated tools to get the same complexity. Then we focus on pattern matching in a sequence of snippets. The idea is to simulate the well-known Knuth-Morris-Pratt algorithm while operating on whole snippets instead of single characters using Lemma 5 and Lemma 6.

▶ **Lemma 5.** *Given a prefix snippet and a suffix snippet we can detect an occurrence of the pattern in their concatenation in constant time.*

▶ **Lemma 6.** *Given a prefix snippet $p[1\,..\,i]$ and a snippet $p[j\,..\,k]$ we can find the longest long border b of $p[1\,..\,i]$ such that $p[1\,..\,b]p[j\,..\,k]$ is a prefix of the whole p in constant time, where a long border is $b \geq \frac{i}{2}$ such that $p[1\,..\,b] = p[i - b + 1\,..\,i]$.*

During the simulation we might create some new snippets, but they will be always either prefix snippets or *half snippets* of the form $p[\frac{i}{2}\,..\,i]$. All information required to make those snippets extended can be precomputed in a relatively straightforward way using $\mathcal{O}(M)$ time.

The running time of the resulting procedure is as much as $\Theta(n \log M)$, though. To accelerate it we try to detect situations when there is a long snippet near the beginning of the sequence, and apply Lemma 7 and Lemma 8 to quickly process all snippets on its left.

▶ **Lemma 7.** *Given a sequence of extended snippets $s_1 s_2 \ldots s_i$ such that $|s_i| \geq 2 \sum_{j<i} |s_j|$, we can detect an occurrence of p in $s_1 s_2 \ldots s_i$ in time $\mathcal{O}(i)$.*

▶ **Lemma 8.** *Given a sequence of extended snippets $s_1 s_2 \ldots s_i$ such that $|s_i| \geq 2 \sum_{j<i} |s_j|$, we can compute the longest prefix of p which is a suffix of $s_1 s_2 \ldots s_i$ in time $\mathcal{O}(i)$.*

After such modification the algorithm works in linear time, which can be shown by defining a potential function depending just on the lengths of the snippets, see the original paper.

3 Overview of the algorithm

Our goal is to detect an occurrence of a pattern $p[1..M]$ in a given text $t[1..N]$, where p and t are described by a Lempel-Ziv-Welch parse of size m and n, respectively. The difficulty here is rather clear: M might be of order m^2, and hence looking at each possible prefix or suffix of the pattern would imply a quadratic (or higher) total complexity. As most efficient uncompressed pattern algorithms are based on a more or less involved preprocessing concerning all prefixes or suffixes, such quadratic behavior seems difficult to avoid. Nevertheless, we can try to use the following reasoning here: either the pattern is really complicated, and then m is very similar to M, hence we can use the linear compressed pattern matching algorithm sketched in the previous section, or it is in some sense repetitive, and we can hope to speedup the preprocessing by building on this repetitiveness. In this section we give a high level formalization of this intuition.

We will try to process whole codewords at once. To this aim we need the following technical lemma which allows us to compare large chunks of the text (or the pattern) in a single step. It follows from the linear time construction of the so-called *suffix tree of a tree* [17] and constant time LCA queries [2].

▶ **Lemma 9.** *It is possible to preprocess in linear time a LZW parse of a text over an alphabet consisting of integers which can be sorted in linear time so that given any two codewords we can compute their longest common suffix in constant time.*

We defer its proof to Section 6 as it is not really necessary to understand the whole idea. As an obvious corollary, given two codewords we can check if the shorter is a suffix of the longer in constant time.

In the very beginning we reverse both the pattern and the text. This is necessary because the above lemma tells how to compute the longest common suffix, and we would actually like to compute the longest common prefix. The only way we will access the characters of both the pattern and the text is either through computing the longest common prefix of two reversed codewords, or retrieving a specified character of a reversed codeword (which can be performed in constant time using level ancestor queries), hence the input can be safely reversed without worrying that it will make working with it more complicated. We call those reversed codewords *blocks*. Note that all suffixes of a block are valid blocks as well.

We start with classifying all possible patterns into two types. Note that this classification depends on both m (size of the compressed pattern) and n (size of the compressed texts) which might seem a little unintuitive.

▶ **Definition 10.** A *kernel* of the pattern is any (uncompressed) substring of length $n+m$ such that its border is less than $\frac{n+m}{2}$. A kernel decomposition of the pattern is its prefix with period at most $\frac{n+m}{2}$ followed by a kernel.

Note that the distance between two occurrences of such substring must be at least $\frac{n+m}{2}$, and hence a kernel occurs at most $2m$ times in the pattern and $2n$ times in the text. It might happen that there is no kernel, or in other words all relatively short fragments are highly repetitive. In such case the whole pattern turns out to be highly repetitive.

▶ **Lemma 11.** *The pattern either has a kernel decomposition or its period is at most* $\frac{n+m}{2}$. *Moreover, those two situations can be distinguished in linear time, and if a decomposition exists it can be found with the same complexity.*

Proof. We start with decompressing the prefix of length $n+m$. If its period d is at least $\frac{n+m}{2}$, we can return it as a kernel. Otherwise we compute the longest common prefix of the

pattern and the pattern shifted by d characters (or, in other words, we compute how far the period extends in the whole pattern). This can be performed using at most $2(n+m)$ queries described in Lemma 9 (note that being able to check if one block is a prefix of another would be enough to get a linear total complexity here, as we can first identify the longest prefix consisting of whole blocks in both words and then inspect the remaining at most $\min(n,m)$ characters naively). If d is the period of the whole pattern, we are done. Otherwise we identified a substring s of length $n+m$ followed by a character a such that the period of s is at most $d \leq \frac{n+m-1}{2}$ but the period of sa is different (larger). We remove the first character of s and get s'. Let d' be the period of $s'a$. If $d' \geq \frac{n+m}{2}$, $s'a$ is a kernel. Otherwise $d, d' \leq \frac{n+m-1}{2}$ are both periods of s', and hence by Lemma 1 they are both multiplies of the period of s'. Let b be the character such that d is a period of $s'b$ (note that $a \neq b$). Because d is a period of $s'b$, $s'[|s'|+1-d] = b$. Similarly, because d' is a period of $s'a$, $s'[|s'|+1-d'] = a$. Hence $s'[|s'|+1-d] \neq s'[|s'|+1-d']$, and because $(|s'|+1-d)-(|s'|+1-d')$ is a multiple of the period of s' we get a contradiction. Note that the prefix before $s'a$ is periodic with period $d \leq \frac{n+m}{2}$ by the construction. ◀

If the pattern turns out to be repetitive, we try to apply the algorithm described in the preliminaries. The intuition is that while we required a certain preprocessing of the whole pattern, when its period is d it is enough to preprocess just its prefix of length $\mathcal{O}(d)$. This intuition is formalized in Section 4. If the pattern has a kernel, we use it to identify $\mathcal{O}(n)$ potential occurrences, which we then manage to verify efficiently. The verification uses a similar idea to the one from Lemma 9 but unfortunately it turns out that we need to somehow compress the pattern during the verification as to keep the running time linear. The details of this part are given in Section 5.

4 Detecting occurrence of a periodic pattern

If the pattern is periodic, we would like to somehow use this periodicity so that we do not have to preprocess the whole pattern (i.e., build the suffix tree, LCA structure, compute the borders of all prefixes and suffixes, and so on). It seems reasonable that preprocessing just the first few repetitions of the period should be enough. More precisely, we will decompress a sufficiently long prefix of p and compute some of its occurrences inside the text. To compute those occurrences we apply a fairly simple modification of Levered-pattern-matching (see [12]) called Lazy-levered-pattern-matching. The pseudocode of the modified version can be seen below.

First observe that both Lemma 5 and Lemma 7 can be modified in a straightforward way so that we get the leftmost occurrence, if any exists. The original procedure quits as soon as it detects that the pattern occurs. We would like it to proceed so that we get more than one occurrence, though. A naive solution would be to simply continue, but then the following situation could happen: both ℓ and $|s_k|$ are very close to m, the pattern occurs both in the very beginning of $p[1\,..\,\ell]s_k$ and somewhere close to the boundary between the two parts, and the longest suffix of the concatenation which is a prefix of the pattern is very short. Then we would detect just the first occurrence, and for some reasons that will be clear in the proof of Lemma 15 this is not enough. Hence whenever there is an occurrence in the concatenation, we skip just the first half of $p[1\,..\,\ell]$ and continue, see lines 11-14. This is the only change in the algorithm.

While Lazy-levered-pattern-matching is not capable of generating all occurrences in some cases, it will always detect a lot of them, in a certain sense. This is formalized in the following lemma.

Algorithm 1 LAZY-LEVERED-PATTERN-MATCHING$(s_1, s_2, \ldots, s_n)$

1: $\ell \leftarrow$ longest prefix of p ending s_1 ▷ **Lemma 3**
2: $k \leftarrow 2$
3: **while** $k \leq n$ and $\ell + \sum_{i=k}^{n} |s_i| \geq M$ **do**
4: choose $t \geq k$ minimizing $|s_k| + |s_{k+1}| + \ldots + |s_{t-1}| - \frac{|s_t|}{2}$
5: **if** $\ell + |s_k| + |s_{k+1}| + \ldots + |s_{t-1}| \leq \frac{|s_t|}{2}$ **then**
6: output the first occurrence of p in $p[1\,..\,\ell]s_k s_{k+1} \ldots s_t$, if any ▷ **Lemma 7**
7: $\ell \leftarrow$ longest prefix of p ending $p[1\,..\,\ell]s_k s_{k+1} \ldots s_t$ ▷ **Lemma 8**
8: $k \leftarrow t + 1$
9: **else**
10: output the first occurrence of p in $p[1\,..\,\ell]s_k$, if any ▷ **Lemma 5**
11: **if** p occurs in $p[1\,..\,\ell]s_k$ **then**
12: $\ell \leftarrow$ longest prefix of p ending $p[\lceil \frac{\ell}{2} \rceil\,..\,\ell]$
13: **continue**
14: **end if**
15: **if** $p[1\,..\,\ell]s_k$ is a prefix of p **then**
16: $\ell \leftarrow \ell + |s_k|$
17: $k \leftarrow k + 1$
18: **continue**
19: **end if**
20: $b \leftarrow$ longest long border of $p[1\,..\,\ell]$ s.t. $p[1\,..\,b]s_k$ is a prefix of p ▷ **Lemma 6**
21: **if** b is undefined **then**
22: $\ell \leftarrow$ longest prefix of p ending $p[\lceil \frac{\ell}{2} \rceil\,..\,\ell]$
23: **continue**
24: **end if**
25: $\ell \leftarrow b + |s_k|$
26: $k \leftarrow k + 1$
27: **end if**
28: **end while**

▶ **Lemma 12.** *If the pattern of length M occurs starting at the i-th character,* LAZY-LEVERED-PATTERN-MATCHING *detects at least one occurrence starting at the j-th character for some $j \in \{i - \frac{M}{2}, i - \frac{M}{2} + 1, \ldots, i\}$.*

Proof. There are just two places where we can lose a potential occurrence: line 7 and 12. More precisely, it is possible that we output an occurrence and then skip a few others. We would like to prove that the occurrences we skip are quite close to the occurrences we output. We consider the two problematic lines separately.

line 7 s_t is a lever, so $\ell + |s_1| + \ldots + |s_t| \leq \frac{3}{2}M$. Hence the distance between any two occurrences of the pattern inside $p[1\,..\,\ell]s_k s_{k+1} \ldots s_t$ is at most $\frac{M}{2}$. We output the first of them, and so can safely ignore the remaining ones.

line 12 If there is an occurrence, we remove the first half of $p[1\,..\,\ell]$ and might skip some other occurrences starting there. If the first occurrence starts later, we will not skip anything. Otherwise we output the first occurrence starting in $p[1\,..\,\frac{\ell}{2}]$, and if there is any other occurrence starting there, their distance is at most $\frac{\ell}{2} \leq \frac{M}{2}$, hence we can safely ignore the latter.

◀

▶ **Lemma 13.** Lazy-levered-pattern-matching *can be implemented to work in time* $\mathcal{O}(n)$ *and use* $\mathcal{O}(M)$ *additional memory.*

Proof. The proof is almost the same as in [12]. The only difference as far as the running time is concerned is line 12. By removing the first half of $p[1\,..\,\ell]$ we either decrease the current potential by 1 or create a lever and thus can amortize the constant time used to locate the first occurrence of the pattern inside $p[1\,..\,\ell]s_k$. ◀

Note that Lazy-levered-pattern-matching works with a sequence of snippets. By first applying the preprocessing mentioned in the preliminaries we can use it to compute a small set which approximates all occurrences in a compressed text.

▶ **Lemma 14.** Lazy-levered-pattern-matching *can be used to compute a set* S *of* $\mathcal{O}(n)$ *occurrences of an uncompressed pattern of length* $M \geq n$ *in a compressed text such that whenever there is an occurrence starting at the* i*-th character,* S *contains* j *from* $\{i - \frac{M}{2}, i - \frac{M}{2} + 1, \ldots, i\}$.

Proof. As mentioned in the preliminaries, we can reduce compressed pattern matching to pattern matching in a sequence of snippets in linear time. Because $M \geq n$, the preprocessing does not produce any occurrences yet. Then we apply Lazy-levered-pattern-matching. Because its running time is linear by Lemma 13, it cannot find more than $\mathcal{O}(n + M)$ occurrences. A closer look at the analysis shows that the number of occurrences produced can be bounded by the potentials of all sequences created during the initial preprocessing phase, which as shown in [12] is at most $\mathcal{O}(n)$. ◀

Now it turns out that if the pattern is compressed but highly periodic, the occurrences found in linear time by the above lemma applied to a sufficiently long prefix of p are enough to detect an occurrence of the whole pattern.

▶ **Lemma 15.** *Fully compressed pattern matching can be solved in linear time if the period of the pattern is at most* $\frac{n+m}{2}$*. Furthermore, given a set of* r *potential occurrences we can verify all of them in* $\mathcal{O}(n + m + r)$ *time.*

Proof. We build the shortest prefix $p[1\,..\,\alpha d]$ of the pattern such that $\alpha d \geq n + m$, where $d \leq \frac{n+m}{2}$ is the period of the whole pattern. Observe that $\alpha d \leq \frac{3}{2}(n + m)$ and hence we can afford to store this prefix in an uncompressed form. By Lemma 14 we construct a set S of $\mathcal{O}(n)$ occurrences of $p[1\,..\,\alpha d]$ such that for any other occurrence starting at the i-th character there exists $j \in S$ such that $0 \leq i - j \leq \frac{\alpha d}{2} \leq \frac{3}{2}(n + m)$. We partition the elements in S according to their remainders modulo d so that $S_t = \{j \in S : j \equiv t \pmod{d}\}$ and consider each S_t separately. Note that we can easily ensure that its elements are sorted by either applying radix sort to the whole S or simply observing that Lazy-levered-pattern-matching generate the occurrences from left to right.

We split S_t into maximal groups of consecutive elements $x_1 < x_2 < \ldots < x_k$ such that $x_{i+1} \leq x_i + \frac{\alpha d}{2}$, which clearly can be performed in linear time with a single left-to-right sweep. Each such group actually corresponds to a fragment starting at the x_1-th character and ending at the $(x_k + \alpha d - 1)$-th character which is a power of $p[1\,..\,d]$. This is almost enough to detect an occurrence of the whole pattern. If the fragment is sufficiently long, we get an occurrence. In some cases this is not enough to detect the occurrence because we might be required to extend the period to the right as to make sufficient space for the whole pattern. Fortunately, it is impossible to repeat $p[1\,..\,d]$ more than $\frac{3}{2}\alpha$ times starting at the x_k character, as otherwise we would have another $x_{k+1} \in S_t$ which we might have used

to extend the group. Hence to compute how far the period extends it would be enough to align $p[1\,..\,\alpha d]p[1\,..\,\frac{\alpha d}{2}]$ starting at the x_k character and compute the first mismatch with the text. We can assume that all suffixes of $p[1\,..\,\alpha d]$ are blocks with just a linear increase in the problem size, and hence we can apply Lemma 9 to preprocess the input so that each such alignment can be processed in time proportional to the number of block in the corresponding fragment of the text. To finish the proof, note that any single block in the text will be processed at most twice. Otherwise we would have two groups ending at the x_k-th and $x'_{k'}$-th characters such that $|x_k + \alpha d - (x'_{k'} + \alpha d)| \leq \frac{\alpha d}{2}$ and that would mean that one of those groups is not maximal. After computing how far the period extends after each group, we only have to check a simple arithmetic conditions to find out if the pattern occurs starting at the corresponding x_1.

To verify a set of r potential occurrences, we construct the groups and compute how far the period extends after each of them as above. Then for each potential occurrence starting at the b_i-th character we lookup the corresponding $S_{b_i \bmod d}$ and find the rightmost group such that $x_1 \leq b_i$. We can verify an occurrence by looking up how far the period extends after the x_k-th character and checking a simple arithmetic condition. To get the claimed time bound, observe that we do not have to perform the lookup separately for each possible occurrence. By first splitting them according to their remainders modulo d and sorting all x_1 and b_i in linear time using radix sort consisting of two passes we get a linear total complexity. ◀

5 Using kernel to accelerate pattern matching

We start with computing all occurrences of the kernel in both the pattern and the text. Because the kernel is long and aperiodic, there are no more than $2m$ of the former and $2n$ of the latter. The question is if we are able to detect all those occurrences efficiently. It turns out that because the kernel is aperiodic, Lazy-levered-pattern-matching can be (again) used for the task. More formally, we have the following lemma.

▶ **Lemma 16.** Lazy-levered-pattern-matching *can be used to compute in* $\mathcal{O}(n + M)$ *time all occurrences of an aperiodic pattern of length* $M \geq n$ *in a compressed text.*

Proof. By Lemma 14 we can construct in linear time a set of occurrences such that any other occurrence is quite close to one of them. But the pattern is aperiodic, so if it occurs at positions i and j with $|i - j| \leq \frac{M}{2}$, then in fact $i = j$. Hence the set contains all occurrences. ◀

We apply the above lemma to find the occurrences of the kernel in both the pattern and the text. Each occurrence of the kernel in the text gives us a possible candidate for an occurrence of the whole pattern (for example by aligning it with the first occurrence of the kernel in the pattern). Hence we have just a linear number of candidates to verify. Still, the verification is not trivial. An obvious approach would be to repeat a computation similar to the one from Lemma 11 for each candidate. This would be too slow, though, as it might turn out that some blocks from the pattern are inspected multiple times. We require a slightly more sophisticated approach.

Using (any) kernel decomposition of the pattern we represent it as $p = p_1p_2p_3$, where the period of p_1 is at most $\frac{n+m}{2}$, and p_2 is a kernel. We start with locating all occurrences of p_2p_3 in the text. It turns out that because p_2 is aperiodic, there cannot be too many of them. Hence we can afford to generate all such occurrences and then verify if any of them is preceded by p_1 as follows:

Algorithm 2 PREF($T[1..|T|]$)

```
1: PREF[1] = 0, s ← 1
2: for i = 2, 3, ..., |T| do
3:     k ← i − s + 1
4:     r ← s + PREF[s] − 1
5:     if r < i then
6:         PREF[i] = NAIVE-SCAN(i, 1)
7:         if PREF[i] > 0 then
8:             s ← i
9:         end if
10:    else if PREF[k] + k < PREF[s] then
11:        PREF[i] ← PREF[k]
12:    else
13:        x ← NAIVE-SCAN(r + 1, r − i + 2)
14:        PREF[i] ← r − i + 1 + x
15:        s ← i
16:    end if
17: end for
18: PREF[1] = |T|
```

1. if $|p_1| \geq n$ then we can directly apply Lemma 15,
2. if $|p_1| < n$ then take the prefix of p_1p_2 consisting of the first $n+m$ letters. Depending on whether this prefix is periodic with the period at most $\frac{n+m}{2}$ or aperiodic, we can apply Lemma 15 or Lemma 16.

The most involved part is computing all occurrences of p_2p_3. To find them we construct a new string $T = p_2p_3\$t[1..N]$ by concatenating the suffix of the pattern and the text. For this new string we compute the values of the prefix function defined in the following way:

$$\text{PREF}[i] = \max\{j : T[k] = T[i+k-1] \text{ for all } k = 1, 2, \ldots, j\}$$

Of course we cannot afford to compute PREF[i] for all possible $N+M$ values of i. Fortunately, PREF[i] $\geq |p_2|$ iff p_2 occurs in T starting at the i-th character. Because $|p_2| = n+m$ and p_2 is aperiodic, there are no more than $2\frac{N+M}{n+m} \leq n+m$ such values of i. We aim to compute PREF[i] just for those i. First lets take a look at the relatively well-known algorithm which computes all PREF[i] for all i, which can be found in the classic stringology book by Crochemore and Rytter [7]. We state its code for the sake of completeness. NAIVE-SCAN(x, y) performs a naive scanning of the input starting at the x-th and y-th characters and returns the first mismatch, if any. PREF uses this procedure in a clever way as to reuse already processed parts of the input and keep the total running time linear. The complexity is linear because the value of $s + \text{PREF}[s]$ cannot decrease nor exceed $|T|$, and whenever it increases we are able to pay for the time spent in NAIVE-SCAN using the difference between the new and the old value.

We will transform this algorithm so that it computes only PREF[i] such that the kernel occurs starting at the i-th character. We call such positions i *interesting*. The first problem we encounter is that we need a constant time access to any PREF[i] and cannot afford to allocate a table of length $|T|$. This can be easily overcome.

▶ **Lemma 17.** *A table* PREF *such that* PREF$[i] > 0$ *iff the kernel occurs starting at the* i*-th character and any entry can be accessed in constant time can be implemented in space and after preprocessing not exceeding the compressed size of* T*, which is* $\mathcal{O}(n+m)$*.*

Proof. Observe that any two occurrences of the kernel cannot be too close. More precisely, their distance must be at least $\frac{n+m}{2}$. We split the whole T into disjoint fragments of size $\frac{n+m}{2}$. There are no more than $2(n+m)$ of them and there is at most one occurrence in each of them. Hence we can implement the table by allocating an array of size $2(n+m)$ with each entry storing at most one element. ◀

We modify line 2 so that it iterates only through interesting values of i. Note that whenever we access some PREF$[j]$ inside, j is either i, s or $k = i - s + 1$. In the first two cases it is clear that the corresponding positions are interesting so we can access the corresponding value using Lemma 17. The third case is not that obvious, though. It might happen that k is not interesting and we will get PREF$[k] = 0$ instead of the true value. If $r \geq i + |p_2| - 1$ then because p_2 occurs at i, it occurs at k as well, and so k is interesting. Otherwise we cannot access the true value of PREF$[k]$, so we start a naive scan by calling Naive-scan$(i + |p_2|, |p_2| + 1)$ (we can start at the $|p_2| + 1$-th character because p_2 occurs at i). After the scanning we set $s \leftarrow i$. Note that because $r < i + |p_2| - 1$, this increases the current value of $s +$ PREF$[s]$, and we can use the increase to amortize the scanning.

We still have to show how to modify Naive-scan. Clearly we cannot afford to perform the comparisons character by character. By the increasing $s +$ PREF$[s]$ argument, any single character from the text is inspected at most once by accessing $T[x]$ (we call it a left side access). It might be inspected multiple times by accessing $T[y]$, though (which we call a right side access). We would like to perform the comparisons block by block using Lemma 9. After a single query we skip at least one block. If we skip a block responsible for the left side access, we can clearly afford to pay for the comparison. We need to somehow amortize the situation when we skip a block responsible for the right side access. For this we will iteratively compress the input (this is similar to the idea used in [10] with the exception that we work with PREF instead of the failure function). More formally, consider the sequence of blocks describing p_2p_3. First note that no further blocks from T will be responsible for a right side access because of the unique \$ character. Whenever some two neighboring blocks b_1, b_2 from this prefix occur in the same block b' from the text, we would like to glue them, i.e., replace by a single block. We cannot be sure that there exists a block corresponding to their concatenation, but because we know where it occurs in b' we can extract (in constant time, by using the level ancestor data structure [3] to preprocess the whole code trie) a block for which the concatenation is a prefix. We will perform such replacement whenever possible. Unfortunately, after such replacement we face a new problem: p_2p_3 is represented as a concatenation of prefixes of blocks instead of whole blocks. Nevertheless, we can still apply Lemma 9 to compute the longest common prefix of two prefixes of blocks $b[1..i]$ and $b'[1..i']$ by first computing the longest common prefix of b and b', and decreasing it if it exceeds $\min(i, i')$. More formally, we store a *block cover* of p_2p_3.

▶ **Definition 18.** A block cover of a word w is a sequence $b_1[1\,..\,i_1], b_2[1\,..\,i_2], \ldots, b_k[1\,..\,i_k]$ of prefixes of blocks such that their concatenation is equal to w.

This definition is illustrated on Figure 1. Obviously, the initial partition of p_2p_3 into blocks is a valid block cover. If during the execution of Naive-scan we find out that two neighboring elements $b_j[1\,..\,i_1], b_{j+1}[1\,..\,i_{j+1}]$ of the current cover occur in some other longer block b, we replace them with the corresponding prefix of b', see Figure 2.

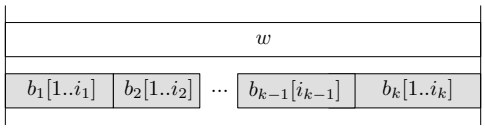

Figure 1 A block cover of w.

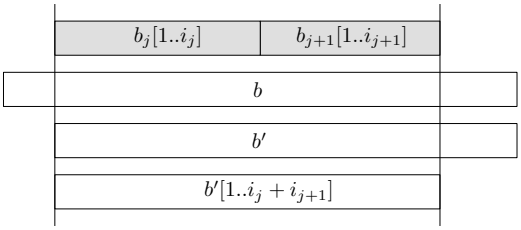

Figure 2 Compressing the current block cover.

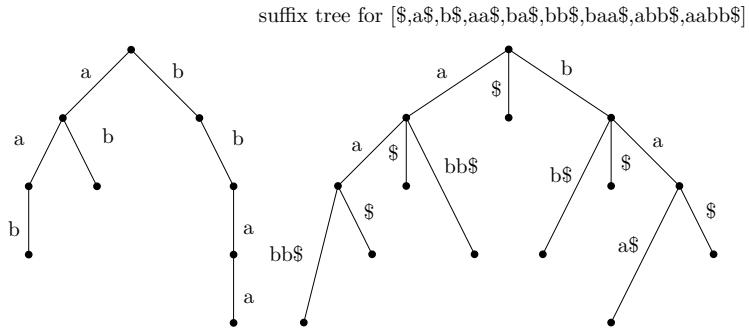

Figure 3 A trie (on the left) and its suffix tree (on the right).

We store all $b_j[1\,..\,i_j]$ on a doubly linked list and update its element accordingly after each replacement. The final required step is to show how we can quickly access in line 13 the block corresponding to $r - i + 2$. We clearly are allowed to spend just constant time there. We keep a pointer to the block covering the r-th character of the pattern. Whenever we need to access the block covering the $(r - i + 2)$-th character, we simply move the pointer to the left, and whenever the current longest match extends, we move the pointer to the right. We cannot move to the left more time than we move to the right, and the latter can be bounded by the number of blocks in the whole p_1p_2 if we replace the neighboring blocks whenever it is possible.

▶ **Lemma 19.** *Fully compressed pattern matching can be solved in linear time if we are given the kernel of the pattern.*

▶ **Theorem 20.** *Fully LZW-compressed pattern matching for strings over a polynomial size integer alphabet can be solved in optimal linear time assuming the word RAM model.*

6 LZW parse preprocessing

The goal of this section is to prove Lemma 9. We aim to preprocess the codewords trie so that given any two codewords, we can compute their longest common suffix in constant time. The *suffix tree of a tree* A, where A is a tree with edges labeled with single characters, is defined as the compressed trie containing $s_A(v)\$$ for all $v \in A$, where $s_A(v)$ is the string constructed by concatenating the labels of all edges on the v-to-root path in A, see Figure 3. This has been first used by Kosaraju [15], who developed a $\mathcal{O}(|A| \log |A|)$ time construction algorithm, where $|A|$ is the number of nodes of A. The complexity has been then improved by Breslauer [6] to just $\mathcal{O}(|A| \log |\Sigma|)$ and by Shibuya [17] to linear for integer alphabets.

We build the suffix tree of the codeword trie T in linear time [17]. As a result we also get for any node v of the input trie the node of the suffix tree corresponding to $s_T(v)\$$.

Now assume that we would like to compute the longest common suffix of two codewords corresponding to nodes u and v in the input trie. In other words, we would like to compute the longest common prefix of $s_T(u)$ and $s_T(v)$. This can be found in constant time after a linear time preprocessing by retrieving the lowest common ancestor of their corresponding nodes in the suffix tree [2].

References

1. Amihood Amir, Gary Benson, and Martin Farach. Let sleeping files lie: pattern matching in z-compressed files. In *SODA '94: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 705–714, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.
2. Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN '00: Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, pages 88–94, London, UK, 2000. Springer-Verlag.
3. Michael A. Bender and Martín Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004.
4. Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
5. Dany Breslauer. Saving comparisons in the Crochemore-Perrin string matching algorithm. In *In Proc. of 1st European Symp. on Algorithms*, pages 61–72, 1995.
6. Dany Breslauer. The suffix tree of a tree and minimizing sequential transducers. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, CPM '96, pages 116–129, London, UK, 1996. Springer-Verlag.
7. Maxime Crochemore and Wojciech Rytter. *Jewels of stringology.* World Scientific, 2002.
8. Zvi Galil. String matching in real time. *J. ACM*, 28(1):134–149, 1981.
9. Zvi Galil and Joel Seiferas. Time-space-optimal string matching (preliminary report). In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 106–113, New York, NY, USA, 1981. ACM.
10. Leszek Gasieniec, Marek Karpinski, Wojciech Plandowski, and Wojciech Rytter. Efficient algorithms for Lempel-Ziv encoding (extended abstract). In *SWAT*, pages 392–403, 1996.
11. Leszek Gasieniec and Wojciech Rytter. Almost optimal fully LZW-compressed pattern matching. In *DCC '99: Proceedings of the Conference on Data Compression*, page 316, Washington, DC, USA, 1999. IEEE Computer Society.
12. Pawel Gawrychowski. Optimal pattern matching in LZW compressed strings. In *SODA '11: Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete algorithms*, pages 362–372, 2011.
13. Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.
14. S. Rao Kosaraju. Pattern matching in compressed texts. In *Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 349–362, London, UK, 1995. Springer-Verlag.
15. S.R. Kosaraju. Efficient tree pattern matching. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:178–183, 1989.
16. James H. Morris, Jr. and Vaughan R. Pratt. A linear pattern-matching algorithm. Technical Report 40, University of California, Berkeley, 1970.
17. Tetsuo Shibuya. Constructing the suffix tree of a tree with a large alphabet. In *Proceedings of the 10th International Symposium on Algorithms and Computation*, ISAAC '99, pages 225–236, London, UK, 1999. Springer-Verlag.